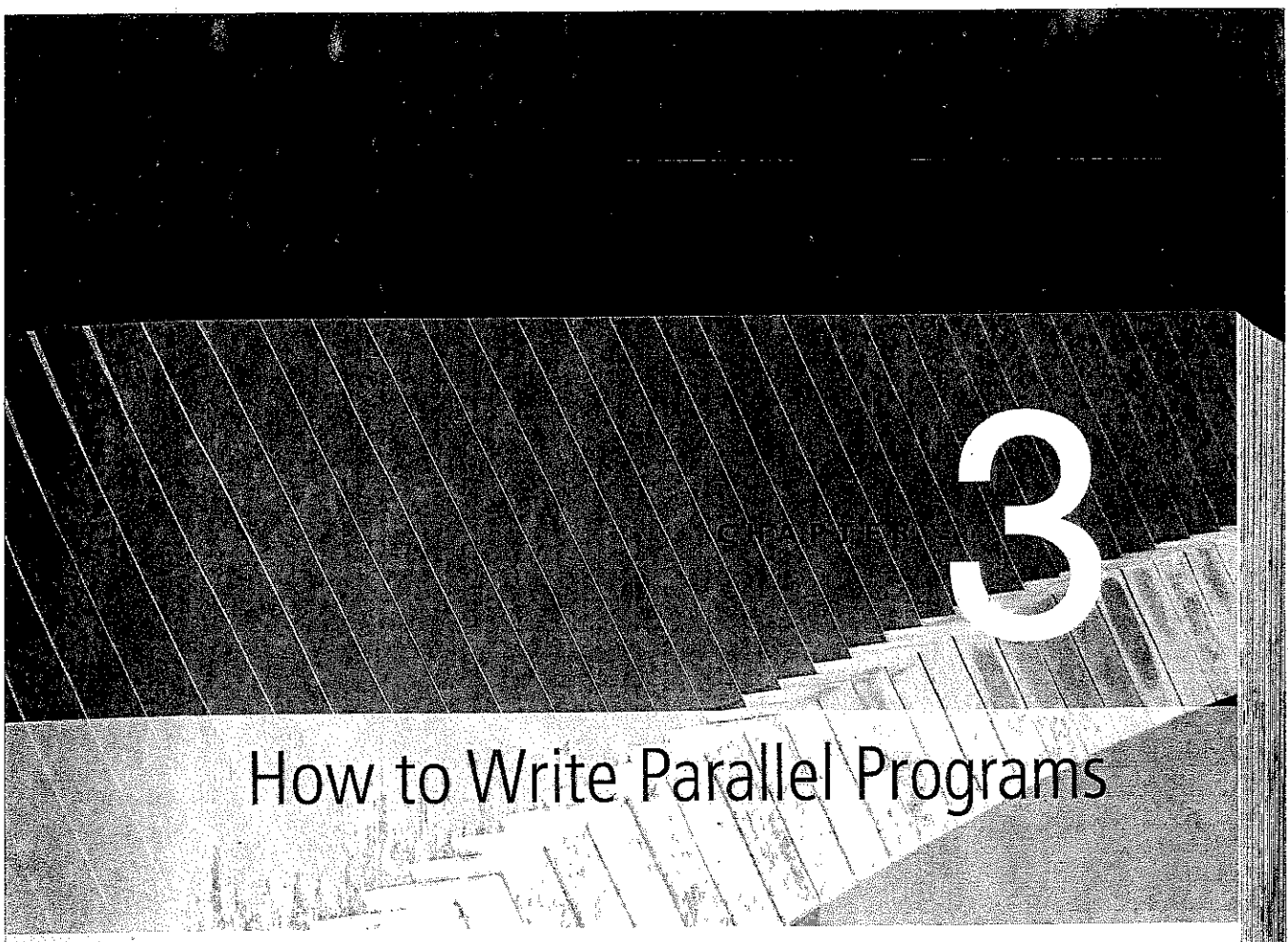
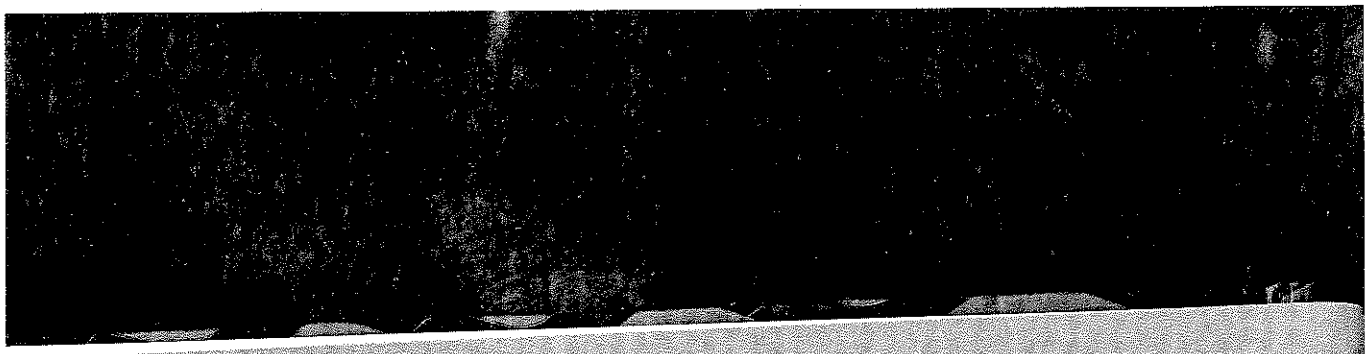


CHAPTER 3

How to Write Parallel Programs

in which we discover the three principal patterns for designing parallel programs; we encounter examples of problems for which each pattern is suited; and we see how to realize the patterns on practical parallel computers



CHAPTER 3 How to Write Parallel Programs

3.1 Patterns of Parallelism

Let's say you are given a problem that requires a massive amount of computation—such as finding, in an enormous genomic database of DNA sequences, the few sequences that best match a short query sequence; or calculating the three-dimensional positions as a function of time of a thousand stars in a star cluster under the influence of their mutual gravitational forces. To get the answer in an acceptable amount of time, you need to write a parallel program to solve the problem. But where do you start? How do you even think about designing a parallel program?

In an 1989 paper titled "How to write parallel programs: a guide to the perplexed," Nicholas Carriero and David Gelernter of Yale University addressed this question by codifying three patterns for designing parallel programs: **result parallelism**; **agenda parallelism**; and **specialist parallelism**. Each pattern encompasses a whole class of similarly structured problems; furthermore, each pattern suggests how to design a parallel program for any problem in that class. Using the patterns, the steps for designing a parallel program are the following:

- Identify the pattern that best matches the problem.
- Take the pattern's suggested design as the starting point.
- Implement the design using the appropriate constructs in a parallel programming language.

Next, we describe each of the three patterns and give examples of how they are used.

3.2 Result Parallelism

In a problem that exhibits **result parallelism** (Figure 3.1), there is a collection of multiple results. The individual results are all computed in parallel, each by its own processor. Each processor is able to carry out the complete computation to produce one result. The conceptual parallel program design is:

Processor 1: Compute result 1

Processor 2: Compute result 2

...

Processor N: Compute result N

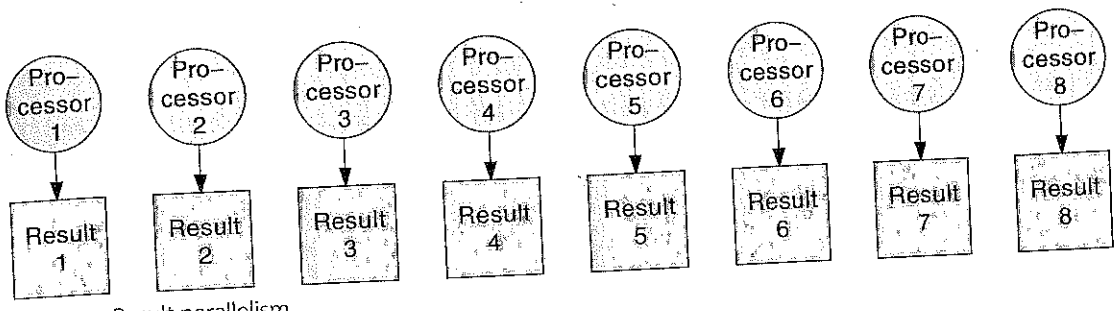


Figure 3.1 Result parallelism

A problem that requires computing each element of a data structure often exhibits result parallelism. As an example, consider the problem of calculating all the pixels in all the frames of a computer-animated film. One way to solve the problem is to assign a separate processor to calculate each pixel. Another way is to assign a separate processor to render each entire frame, calculating all the pixels in that frame. The former is an example of **fine-grained parallelism**, where each result requires a small amount of computation. The latter is an example of **coarse-grained parallelism**, where each result requires a large amount of computation. In both cases, though, none of the processors needs to use the result calculated by any other processor; there are no dependencies among the computations. Thus, in concept, all the processors can start at the same time, calculate, and finish at the same time.

Other problems, however, do have dependencies among the computations. Consider calculating the 3-D positions of N stars in a star cluster as a function of time for a series of M time steps. The result can be viewed as an $M \times N$-element matrix of positions: row 1 contains the stars' positions after the first time step; row 2 contains the stars' positions after the second time step; and so on. In concept, the parallel program has $M \times N$ processors. The processors in row 1 can begin computing their results immediately. Each processor in row 1 calculates the gravitational force on its star due to each of the other stars, using the stars' input initial positions. Each processor in row 1 then moves its star by one time step in a direction determined by the net gravitational force, and the star's new position becomes the processor's result. However, the processors in row 2 cannot begin computing their results immediately. To do their computations, these processors need the stars' positions after the first time step, so these processors must wait until all the row-1 processors have computed their results. We say there are **sequential dependencies** from the computations in each row to the computations in the next row (Figure 3.2). There are no sequential dependencies between the computations in the same row, though.

Faced with the problem of calculating stellar motion for, say, one thousand stars and one million time steps, you might wonder where to find a parallel computer with enough hardware to compute each element of the result in its own separate processor. Keep in mind that, for now, we are still in the realm of *conceptual* parallel program design, where there are no pesky hardware limitations. In Section 3.5 we will see how to translate this conceptual design into a real parallel program.

Recalculating a spreadsheet is another example of a result parallel problem with sequential dependencies. The spreadsheet cell values are the results computed by the program. Conceptually, each cell has its own processor that computes the value of the cell's formula. When you type a new value into an input cell, the processors all calculate their respective cells' formulas. Normally, all the cells can be calculated in parallel. However, if the formula for cell B1 uses the value of cell A1, then the B1 processor must wait until the A1 processor has finished. Soon all spreadsheets will have to be result parallel programs to get full performance out of a desktop or laptop computer's multicore CPU chip.

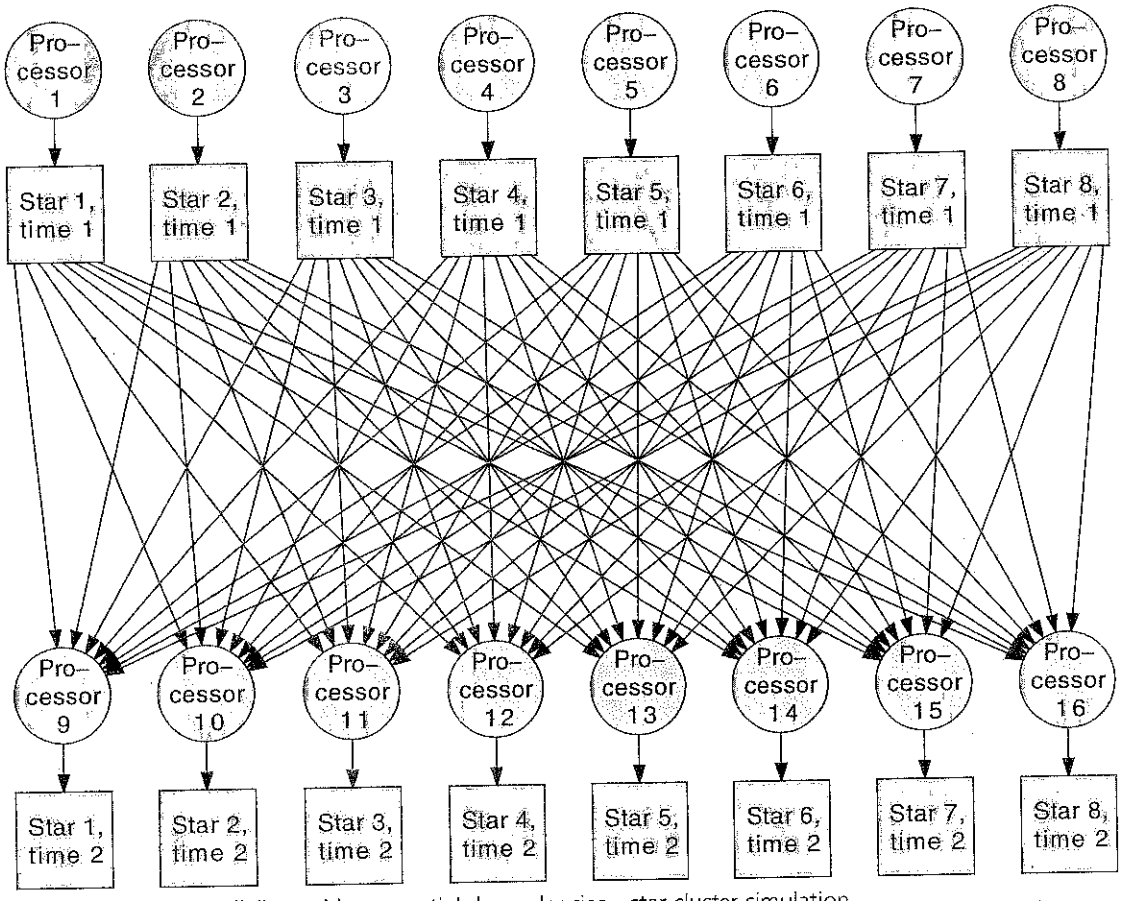


Figure 3.2 Result parallelism with sequential dependencies—star cluster simulation

3.3 Agenda Parallelism

In a problem that exhibits **agenda parallelism** (Figure 3.3), there is an agenda of tasks that must be performed to solve the problem, and there is a team of processors, each processor able to perform any task. The conceptual parallel program design is:

Processor 1: Perform task 1

Processor 2: Perform task 2

. . .

Processor *N*: Perform task *N*

A problem that requires computing one result, or a small number of results, from a large number of inputs often exhibits agenda parallelism. Querying a DNA sequence database is one example. The agenda items are: "Determine if the query sequence matches database sequence 1"; "Determine if the query sequence matches database sequence 2"; and so on. Each of these tasks can be performed independently of all the others, in parallel.

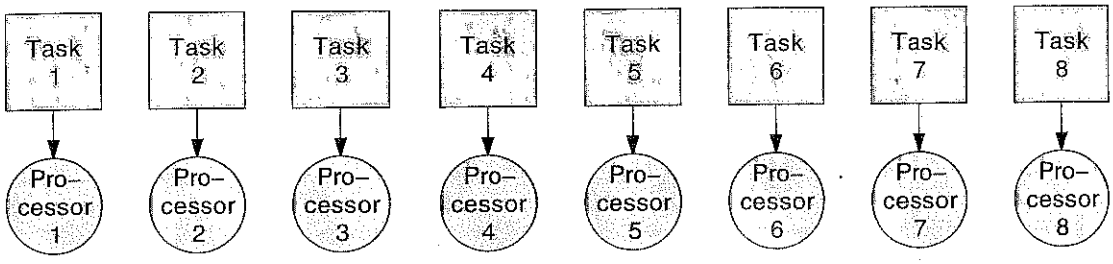


Figure 3.3 Agenda parallelism

Other agenda parallel problems have sequential dependencies among the tasks (Figure 3.4). Certain tasks cannot start until other tasks have finished. The Basic Local Alignment Search Tool (BLAST) program, a widely used DNA and protein sequence database search program, can be viewed as an agenda parallel problem. BLAST proceeds in a series of phases. In the first phase, BLAST looks for matches between short pieces of the query sequence and short pieces of the sequence database; this results in a large number of tentative starting points known as "seeds." In the second phase, BLAST takes each seed and tries to align the complete query sequence with the sequence database, starting from the seed's location. BLAST computes a score for each alignment that tells how biologically plausible the alignment is; alignments that don't result in a good match (too low a score) are discarded. In the third phase, BLAST sorts the surviving alignments into descending order of plausibility and outputs the alignments, most plausible first. Conceptually, the phase-1 agenda items are of the form "For seed X, match piece Y of the query against piece Z of the database." These can all be done in parallel. The phase-2 agenda items are of the form "Align the query with the database at seed X's location and compute the plausibility score." These can all be done in parallel with each other, but each must wait until the corresponding phase-1 agenda item has finished. The final agenda item, "Sort and output the alignments," must wait until the phase-2 agenda items have finished.

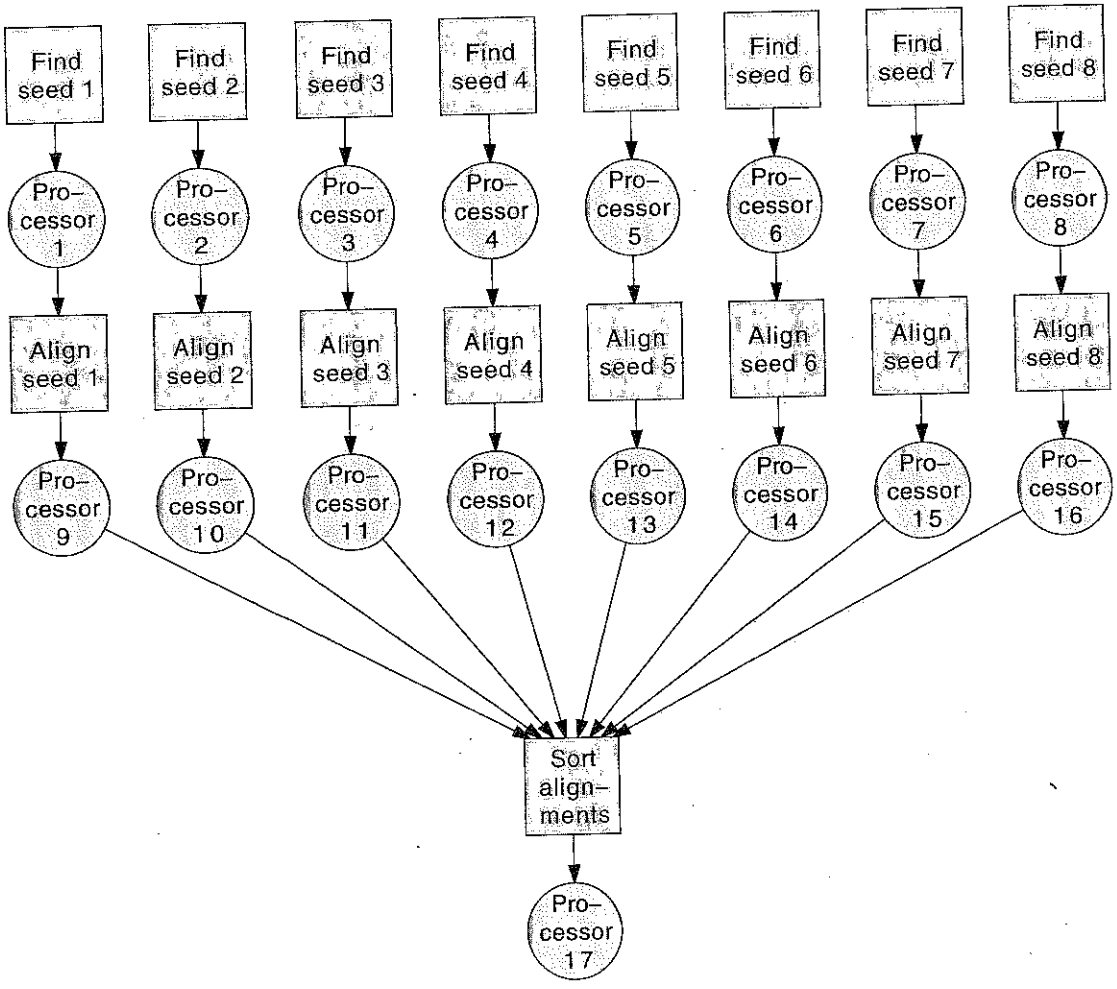


Figure 3.4 Agenda parallelism with sequential dependencies—BLAST

A result parallel problem could be viewed as an agenda parallel problem, where the agenda items are "Compute result 1," "Compute result 2," and so on. The difference is that in a result parallel problem, we are typically interested in *every* processor's result. In an agenda parallel problem, we are typically not interested in every processor's (agenda item's) result, but only in certain results, or only in a combination or summary of the individual results.

When an agenda parallel program's output is a combination or summary of the individual tasks' results, the program is following the so-called **reduction** pattern (Figure 3.5). The number of results is *reduced* from many down to one. Often, the final result is computed by applying a **reduction operator** to the individual results. For example, when the operator is addition, the final result is the sum of the tasks' results. When the operator is minimum, the final result is the smallest of the tasks' results.

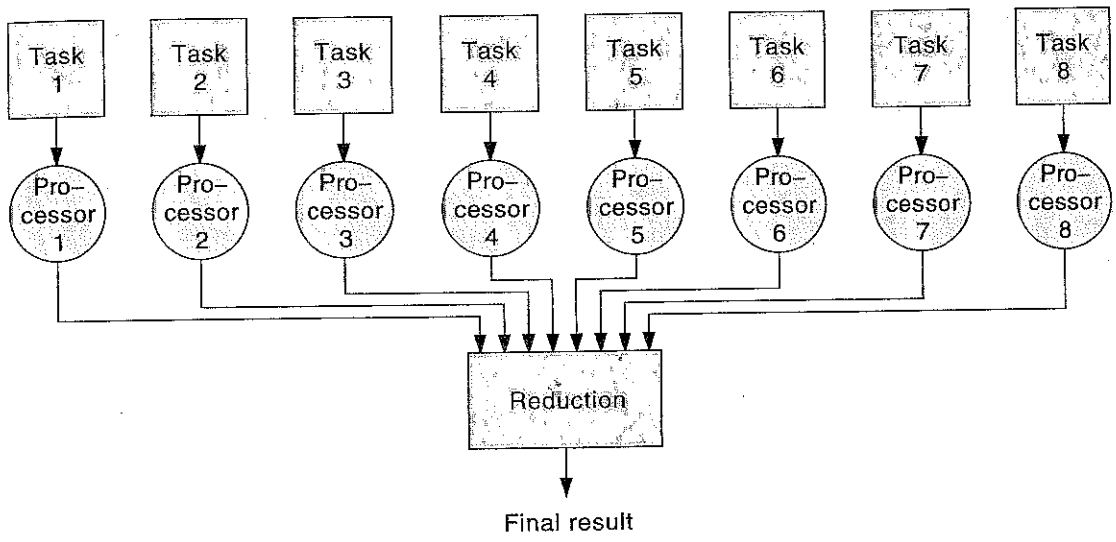


Figure 3.5 Agenda parallelism with reduction

3.4 Specialist Parallelism

In a problem that exhibits **specialist parallelism** (Figure 3.6), like agenda parallelism, there is a group of tasks that must be performed to solve the problem, and there is a team of processors. But, unlike agenda parallelism, each processor performs only a specific one of the tasks, not just any task. Often, one specialist processor's job is to perform the same task on a series of items. The conceptual parallel program design is:

```
Processor 1:    For each item:
                    Perform task 1 on the item

Processor 2:    For each item:
                    Perform task 2 on the item

...

Processor N:    For each item:
                    Perform task N on the item
```

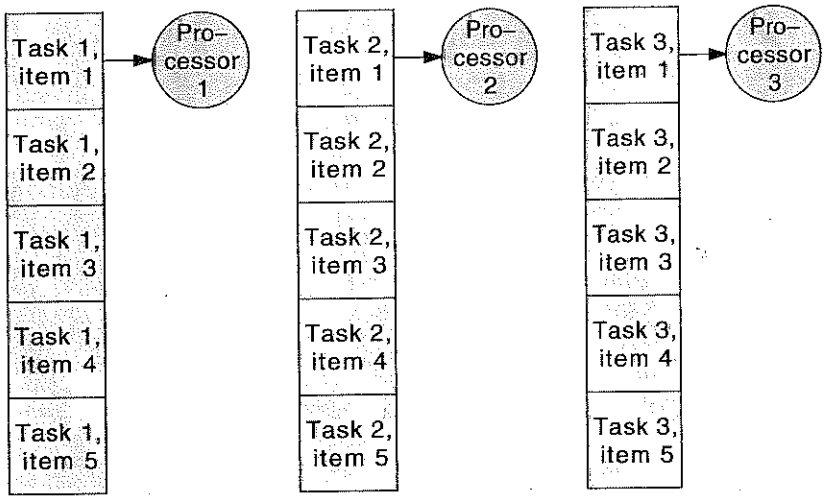


Figure 3.6 Specialist parallelism

When there are sequential dependencies between the tasks in a specialist parallel problem, the program follows the so-called **pipeline** pattern. The output of one processor becomes the input for the next processor. All the processors execute in parallel, each taking its input from the preceding processor's previous output.

Consider again the problem of calculating the 3-D positions of N stars in a star cluster as a function of time for a series of M time steps. Now add a feature: At each time step, the program must create an image of the stars' positions and store the image in a Portable Network Graphics (PNG) file. This problem can be broken into three steps: calculate the stars' positions; create an image of the star's positions; and store the image in a PNG file. Each step requires a certain amount of computation: to calculate the numerical (x,y,z) coordinates of each star; to determine the color of each pixel so as to display the stars' 3-D positions properly in the 2-D image; and to compress the pixel data and store it in a PNG file. The three steps can be performed in parallel by three processors in a specialist parallel program (Figure 3.7). While one processor is calculating the stars' positions for time step t, another processor is taking the stars' positions for time step t–1 and rendering an image, and a third processor is taking the image for time step t–2, compressing it, and storing it in a file. A program like this, where some processors are doing computations and other processors are doing file input or output, is said to be using the **overlapping** pattern, also called the **overlapped computation and I/O** pattern.

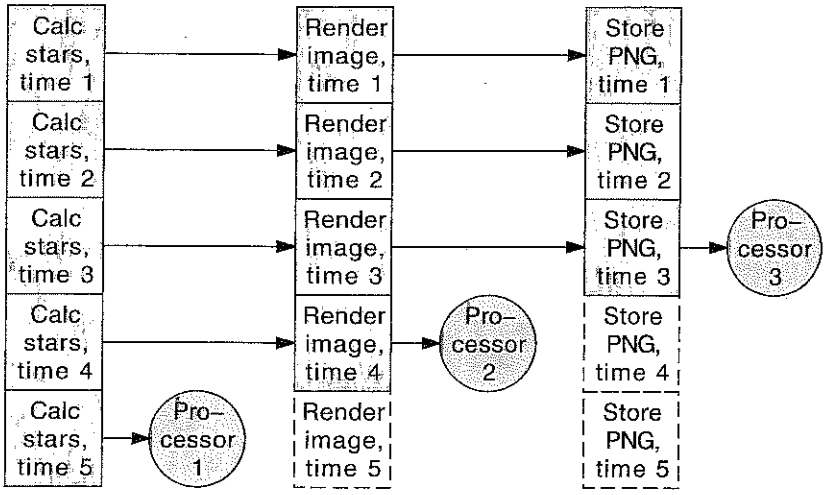


Figure 3.7 Specialist parallelism with sequential dependencies—star cluster simulation

It's possible for a problem to exhibit multiple patterns of parallelism. The star cluster program, for example, can combine result parallelism with specialist parallelism. At each time step, we can have N processors each calculating one star's position (result parallelism); one processor rendering the image for the previous time step (specialist parallelism); and one processor compressing and writing the previous image to a PNG file (specialist parallelism). Putting it another way, the specialist parallel task of computing the stars' positions for one time step is itself a subproblem that exhibits result parallelism.

To sum up the three patterns: Result parallelism focuses on the results that can be computed in parallel. Agenda parallelism focuses on the tasks that can be performed in parallel. Specialist parallelism focuses on the processors that can execute in parallel.

3.5 Clumping, or Slicing

Applying the parallel program design patterns, as described so far, to a problem large enough to need a parallel computer would require a veritable horde of processors. A result parallel problem with a billion results would require a billion processors, one to compute each result. An agenda parallel problem with a billion agenda items would require a billion processors as well. (Specialist parallel problems tend not to require such large numbers of *different* specialists.) A problem size of one billion—10^9, or about 2^{30}—is by no means far-fetched. We will run even the simple, pedagogical parallel programs in this book on problems of this size. Real-world parallel programs regularly run on much larger problems.

The difficulty, of course, is finding a parallel computer with billions and billions of processors. Well-funded government or academic high-performance computing centers may have parallel computers with processors numbering in the thousands. Most of us would count ourselves lucky to have a dozen or two.

To fit a large parallel problem on an actual parallel computer with comparatively few processors, we use **clumping**. Many conceptual processors are clumped together and executed by one actual processor. In a result parallel program, each processor computes a clump of many results instead of just one result (Figure 3.8).

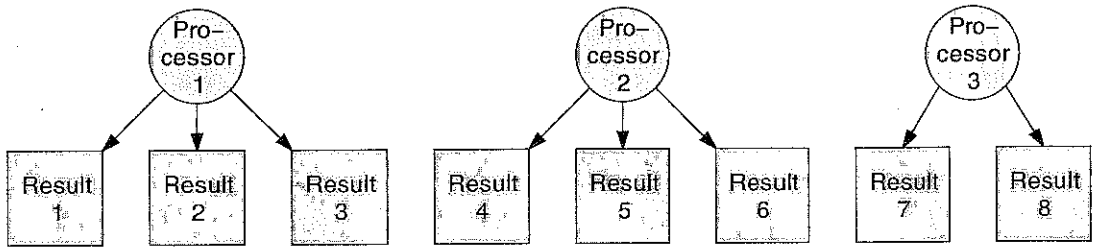


Figure 3.8 Result parallelism with clumping (or slicing)

Slicing is another way of looking at the same thing. Rather than thinking of clumping many processors into one, think of dividing the result data structure into slices, as many slices as there are processors, and assigning one processor to compute all the results in the corresponding slice. For example, suppose there are 100 results and 4 processors. The design of the result parallel program with slicing is:

Processor 1: Compute result 1, 2, . . . 24, 25
Processor 2: Compute result 26, 27, . . . 49, 50
Processor 3: Compute result 51, 52, . . . 74, 75
Processor 4: Compute result 76, 77, . . . 99, 100

In the rest of the book, we will study several parallel programming constructs that automatically slice up a problem of any size to use however many processors the parallel computer has.

3.6 Master-Worker

An agenda parallel problem with many more tasks than processors must also use clumping on a real parallel computer (Figure 3.9). Each processor performs many tasks, not just one. Conceptually, the agenda takes the form of a bag of tasks. Each processor repeatedly takes a task out of the bag and performs the task, until the bag is empty, as follows:

Processor 1: While there are more tasks:
Get and perform the next task
Processor 2: While there are more tasks:
Get and perform the next task
. . .
Processor *K*: While there are more tasks:
Get and perform the next task

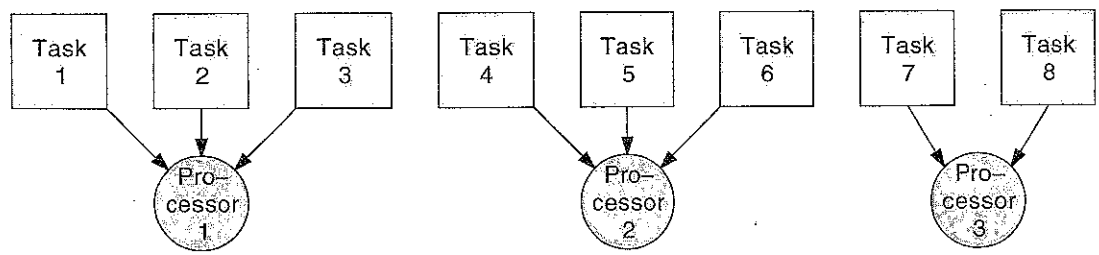


Figure 3.9 Agenda parallelism with clumping

On a cluster parallel computer, an agenda parallel problem with clumping is often realized concretely using the **master-worker** pattern (Figure 3.10). There is one master processor in charge of the agenda, and there are K worker processors that carry out the agenda items. The master sends tasks to the workers, receives the task results from the workers, and keeps track of the program's overall results. Each worker receives tasks from the master, computes the task results, and sends the results back to the master. The conceptual parallel program design is:

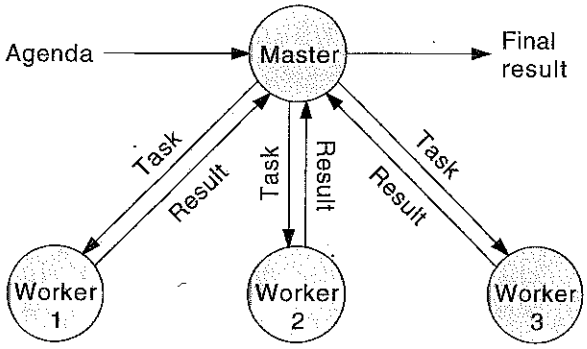


Figure 3.10 Agenda parallelism, master-worker pattern

Master:
    Send initial task to each worker
    Repeat:
        Receive task result from any worker X
        Record task result
        Get next task
        If there are no more tasks, tell worker X to stop
        Otherwise, send task to worker X

Worker 1:
    Repeat:
        Receive a task from the master
        If there are no more tasks, stop
        Compute task results
        Send results to the master

Worker 2:
    Repeat:
        Receive a task from the master
        If there are no more tasks, stop
        Compute task results
        Send results to the master

...

Worker K:
    Repeat:
        Receive a task from the master
        If there are no more tasks, stop
        Compute task results
        Send results to the master

In the rest of the book, we will study many parallel programs designed to run on SMP parallel computers, cluster parallel computers, and hybrid cluster parallel computers. All these programs, however, will follow one of the three parallel design patterns—result parallelism, agenda parallelism, specialist parallelism—or a combination thereof. Before diving into an in-depth study of the parallel programming constructs that let us implement these patterns, in Chapter 4 we will wet our toes with a small introductory parallel program.

3.7 For Further Information

On the three parallel design patterns—Carriero's and Gelernter's paper:

- N. Carriero and D. Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

Carriero and Gelernter later expanded their paper into a book:

- N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

A more recent book about parallel design patterns, coming from the "patterns movement" in software design:

- T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

On the Basic Local Alignment Search Tool (BLAST)—the original paper:

- S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 5, 1990.

Sequential implementations of BLAST:

- FSA-BLAST. http://www.fsa-blast.org/
- NCBI BLAST. http://www.ncbi.nlm.nih.gov/
- WU-BLAST. http://blast.wustl.edu/

Parallel implementations of BLAST:

- mpiBLAST. http://www.mpiblast.org/
- ScalaBLAST. http://hpc.pnl.gov/projects/scalablast/

CHAPTER 4

A First Parallel Program

in which we build a simple sequential program; we convert it to a program for an SMP parallel computer; we see how long it takes to run each version; and we get some insight into how parallel programs execute

CHAPTER 4 A First Parallel Program

4.1 Sequential Program

To demonstrate a program that can benefit from running on a parallel computer, let's invent a simple computation that will take a long time. Here is a Java subroutine that decides whether a number x is prime using the **trial division** algorithm. The subroutine tries to divide x by 2 and by every odd number p from 3 up to the square root of x. If any remainder is 0, then p is a factor of x and x is not prime; otherwise x is prime. While trial division is by no means the fastest way to test primality, it suffices for this demonstration program.

```
private static boolean isPrime
   (long x)
   {
   if (x % 2 == 0) return false;
   long p = 3;
   long psqr = p*p;
   while (psqr <= x)
      {
      if (x % p == 0) return false;
      p += 2;
      psqr = p*p;
      }
   return true;
   }
```

Here is a main program that uses a loop to call the subroutine with the values of x specified on the command line.

```
static int n;
static long[] x;

public static void main
   (String[] args)
   throws Exception
   {
   n = args.length;
```

```
    x = new long [n];
    for (int i = 0; i < n; ++ i)
       {
       x[i] = Long.parseLong (args[i]);
       }
    for (int i = 0; i < n; ++ i)
       {
       isPrime (x[i]);
       }
 }
```

When we run the primality testing program, we want to know the time when each subroutine call starts and the time when each subroutine call finishes, relative to the time the program started. This tells us how long it took to run the subroutine. To measure these times, we use Java's `System.currentTimeMillis()` method, which returns the wall clock time in milliseconds (msec). We record each instant in a variable, and postpone printing the results, so as to disturb the timing as little as possible while the program is running. It can take several msec to call `println()`, and we don't want to include that time in our measurements.

```
static int n;
static long[] x;
static long t1, t2[], t3[];

public static void main
   (String[] args)
   throws Exception
   {
   t1 = System.currentTimeMillis();
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   t2 = new long [n];
   t3 = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      t2[i] = System.currentTimeMillis();
      isPrime (x[i]);
      t3[i] = System.currentTimeMillis();
      }
}
```

Here is the complete Java class, Program1Seq, including code to print the running time measurements.

```
public class Program1Seq
    {
    static int n;
    static long[] x;
    static long t1, t2[], t3[];

  public static void main
    (String[] args)
    throws Exception
    {
    t1 = System.currentTimeMillis();
    n = args.length;
    x = new long [n];
    for (int i = 0; i < n; ++ i)
       {
       x[i] = Long.parseLong (args[i]);
       }
    t2 = new long [n];
    t3 = new long [n];
    for (int i = 0; i < n; ++ i)
       {
       t2[i] = System.currentTimeMillis();
       isPrime (x[i]);
       t3[i] = System.currentTimeMillis();
       }
    for (int i = 0; i < n; ++ i)
       {
       System.out.println
          ("i = "+i+" call start = "+(t2[i]-t1)+" msec");
       System.out.println
          ("i = "+i+" call finish = "+(t3[i]-t1)+" msec");
       }
    }

  private static boolean isPrime
    (long x)
    {
    if (x % 2 == 0) return false;
    long p = 3;
    long psqr = p*p;
    while (psqr <= x)
```

```
        {
        if (x % p == 0) return false;
        p += 2;
        psqr = p*p;
        }
      return true;
      }
}
```

4.2 Running the Sequential Program

When run on an SMP parallel computer, Program1Seq prints the following. The parallel computer has four processors, each a 450 MHz Sun Microsystems UltraSPARC-II CPU, and 2 GB of shared main memory. (Running the program on a different computer would, in general, yield different results.) All four arguments happen to be prime numbers.

```
$ java Program1Seq 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 1 msec
i = 0 call finish = 3842 msec
i = 1 call start = 3842 msec
i = 1 call finish = 7663 msec
i = 2 call start = 7663 msec
i = 2 call finish = 11502 msec
i = 3 call start = 11502 msec
i = 3 call finish = 15342 msec
```

Plotting each subroutine call's start and finish on a timeline reveals how the program executes (Figure 4.1). The program executes each subroutine call in its entirety before going on to the next subroutine call. Because the program's statements are executed in sequence with no overlap in time, we call it a **sequential program**. There is no parallelism, even when running on a parallel computer.

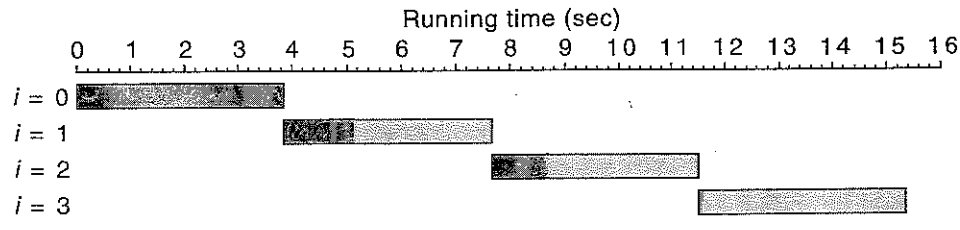


Figure 4.1 Program1Seq execution timeline, SMP parallel computer

4.3 SMP Parallel Program

Now let's rewrite the program using Parallel Java so it will run in parallel when executed on an SMP parallel computer. In the main program, after extracting the command line arguments, we create a **parallel team** object. The constructor argument, `n`, says we want as many threads in the parallel team as there are values to test for primality.

```
public static void main
   (String[] args)
   throws Exception
   {
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   new ParallelTeam(n);
   }
```

Each thread in the parallel team simultaneously executes the code in a **parallel region** object, declared here as an anonymous inner class. The actual parallel code goes in the parallel region's `run()` method.

```
public static void main
   (String[] args)
   throws Exception
   {
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   new ParallelTeam(n).execute (new ParallelRegion()
      {
      public void run()
         {
         }
      });
   }
```

Rather than use a loop to execute the computations (subroutine calls) in sequence, we want the threads to execute the computations in parallel. To make this happen, we put the code for one computation in the parallel region's `run()` method. However, we want each computation to use a different x value. To make this happen, we set `i` to the index of the calling thread within the parallel team (0 through 3, as returned by the parallel region's `getThreadIndex()` method).

```
public static void main
   (String[] args)
   throws Exception
   {
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   new ParallelTeam(n).execute (new ParallelRegion()
      {
      public void run()
         {
         int i = getThreadIndex();
         isPrime (x[i]);
         }
      });
   }
```

Here is the complete Java class, Program1Smp, including code to record the running time measurements and print them after the parallel region has finished executing.

```
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
public class Program1Smp
   {
   static int n;
   static long[] x;
   static long t1, t2[], t3[];

public static void main
   (String[] args)
   throws Exception
```

```
    {
    t1 = System.currentTimeMillis();
    n = args.length;
    x = new long [n];
    for (int i = 0; i < n; ++ i)
       {

       x[i] = Long.parseLong (args[i]);
       }
    t2 = new long [n];
    t3 = new long [n];
    new ParallelTeam(n).execute (new ParallelRegion()
       {
       public void run()
          {
          int i = getThreadIndex();
          t2[i] = System.currentTimeMillis();
          isPrime (x[i]);
          t3[i] = System.currentTimeMillis();
          }
       });
    for (int i = 0; i < n; ++ i)
       {
       System.out.println
          ("i = "+i+" call start = "+(t2[i]-t1)+" msec");
       System.out.println
          ("i = "+i+" call finish = "+(t3[i]-t1)+" msec");
       }
    }

 private static boolean isPrime
    (long x)
    {
    if (x % 2 == 0) return false;
    long p = 3;
    long psqr = p*p;
    while (psqr <= x)
       {
       if (x % p == 0) return false;
       p += 2;
       psqr = p*p;
       }
    return true;
    }
}
```

Here's how the program works (Figure 4.2). The main program begins with one thread, the "main thread," executing the `main()` method. When the main thread creates the parallel team object, the parallel team object creates additional hidden threads; the constructor argument specifies the number of threads. These form a "team" of threads for executing code in parallel. When the main thread calls the parallel region's `execute()` method, the main thread suspends execution and the parallel team threads take over. All the team threads call the parallel region's `run()` method simultaneously, each thread retrieves a value for x, and each thread calls `isPrime()`. Thus, the `isPrime()` subroutine calls happen at the same time, and each subroutine call is performed by a different thread with a different argument. When all the subroutine calls have finished executing, the main thread resumes executing statements after the parallel region and prints the timing measurements.

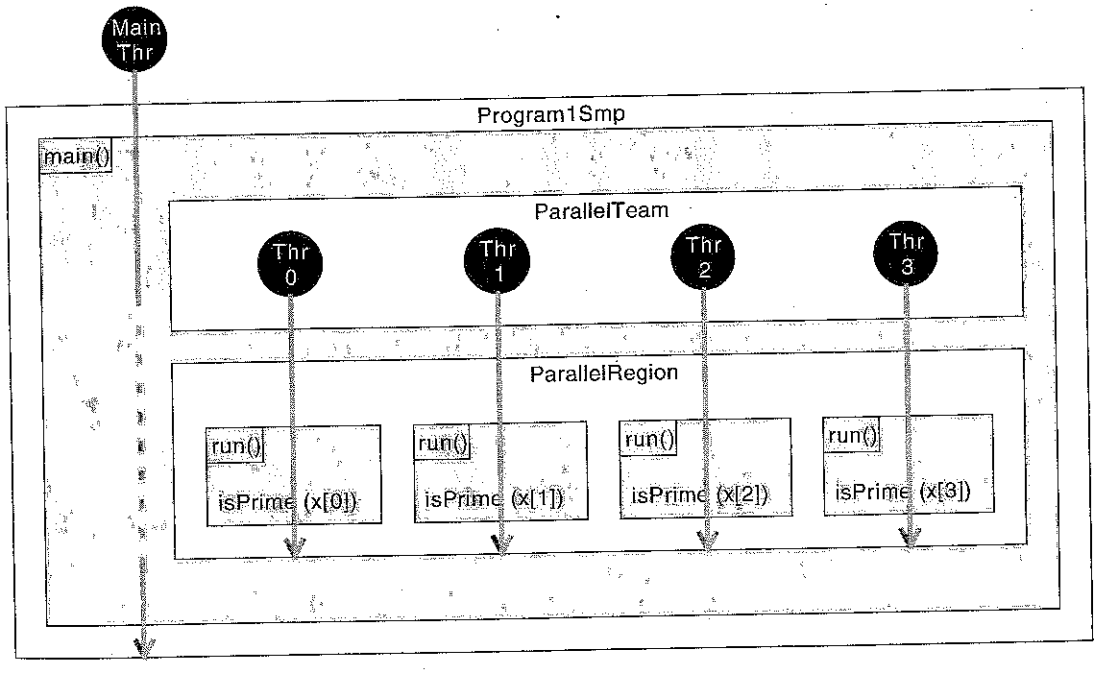


Figure 4.2 Program1Smp operation

When running such a thread-based program on an SMP parallel computer, the Java Virtual Machine (JVM) and the operating system are responsible for scheduling each thread to execute on a different processor. Thus, the computations done by each thread—in this case, the different subroutine calls—are executed in parallel on different processors, resulting in a speedup with respect to the sequential program.

The parallel program illustrates a central theme of parallel program design: *Repetition does not necessarily imply sequencing.* The sequential program used a loop to get n repetitions of a subroutine call. As a side effect, the loop did the repetitions *in sequence.* However, for this program there is no need to do the repetitions in sequence. We wrote the original program with a loop because Java, like many programming languages, only has constructs for expressing a *sequence* of repetitions (a loop). So accustomed are we to this feature that whenever we are confronted with a repeated calculation, we automatically think "loop." However, a loop is not the only way to do a repeated calculation. Provided the

repetitions do not have to be done in sequence, another way to do a repeated calculation is to run several copies of the calculation in multiple threads. A large part of the effort in learning parallel program design is breaking the habit of always using a loop to do repetitions in sequence, and forming the new habit of doing repetitions in parallel whenever possible.

4.4 Running the Parallel Program

When run on the four-processor parallel computer, Program1Smp printed the following:

```
$ java Program1Smp 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 125 msec
i = 0 call finish = 4076 msec
i = 1 call start = 125 msec
i = 1 call finish = 4098 msec
i = 2 call start = 125 msec
i = 2 call finish = 4082 msec
i = 3 call start = 125 msec
i = 3 call finish = 4076 msec
```

Now the timeline (Figure 4.3) shows parallelism. (Compare Figures 4.1 and 4.3 to Figure 1.2.) All the computations start at the same time, execute simultaneously, and finish at about the same time. Whereas the sequential version's running time was 15342 msec, the parallel version's running time on four processors was 4098 msec—a reduction by a factor of about four, as expected.

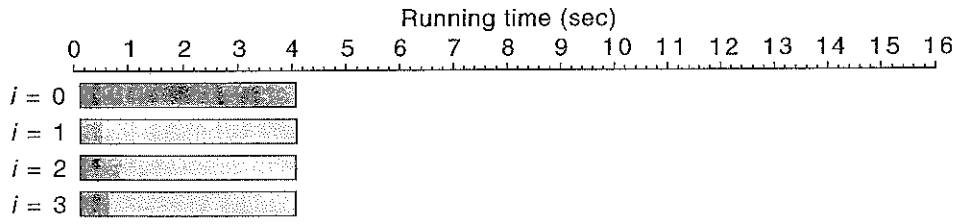


Figure 4.3 Program1Smp execution timeline, SMP parallel computer

To be precise, the speedup (the reduction factor) was 15342/4098 = 3.744. The speedup was somewhat less than 4 because of overhead in the parallel version that is not present in the sequential version. With Program1Smp, the first subroutine call didn't begin until 125 msec after the program started. During this time, the program was occupied in creating the parallel team and parallel region objects, starting up the parallel team threads, and executing the parallel region's `run()` method—work that the sequential program didn't have to do.

This illustrates another central theme of parallel program design: *Parallelism is not free.* The benefit of speedup or sizeup comes with a price of extra overhead that is not needed in a sequential program. The name of the game is to minimize this extra overhead.

4.5 Running on a Regular Computer

Although intended to run on a parallel computer, Program1Seq and Program1Smp are perfectly happy to run on a nonparallel computer. In fact, one benefit of programming in Parallel Java is that you can develop and test parallel programs on any computer, and then you can shift to a parallel computer when the program is debugged and ready for usage.

Let's look at what happens when we run these programs on a regular computer. This was a nonparallel computer with a 1.6 GHz Intel Pentium CPU and 512 MB of main memory. The sequential Program1Seq program printed the following:

```
$ java Program1Seq 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 0 msec
i = 0 call finish = 1594 msec
i = 1 call start = 1594 msec
i = 1 call finish = 2881 msec
i = 2 call start = 2881 msec
i = 2 call finish = 4165 msec
i = 3 call start = 4165 msec
i = 3 call finish = 5450 msec
```

The timeline (Figure 4.4) shows the typical pattern of sequential execution.

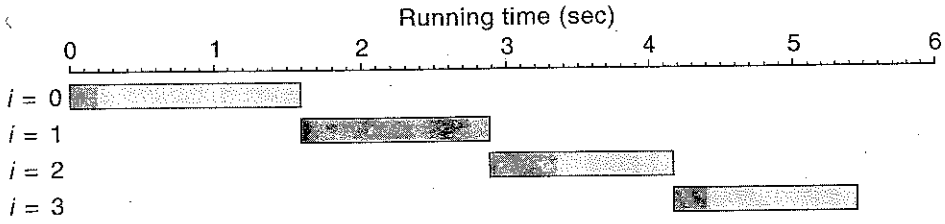


Figure 4.4 Program1Seq execution timeline, regular computer

Suppose we run the multithreaded Program1Smp program on the regular computer. Because each subroutine call will now run in a different thread, we would expect all the subroutine calls to start at roughly the same time near the beginning of the program. But because all the threads will share the same processor, and the processor will execute one thread at a time and switch to another thread every so often, we would expect the overall running time to be about the same as the sequential program. Here is what the parallel program printed.

```
$ java Program1Smp 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 21 msec
i = 0 call finish = 5190 msec
```

```
i = 1 call start = 71 msec
i = 1 call finish = 5053 msec
i = 2 call start = 91 msec
i = 2 call finish = 5134 msec
i = 3 call start = 14 msec
i = 3 call finish = 4981 msec
```

The timeline for this run (Figure 4.5) is about what we expected—except for one thing. The overall running time was 260 msec shorter for the four-thread parallel version than for the single-thread sequential version. We got a slight but noticeable speedup when we went from one thread to four threads on the regular computer. How can this be, when there was only one processor?

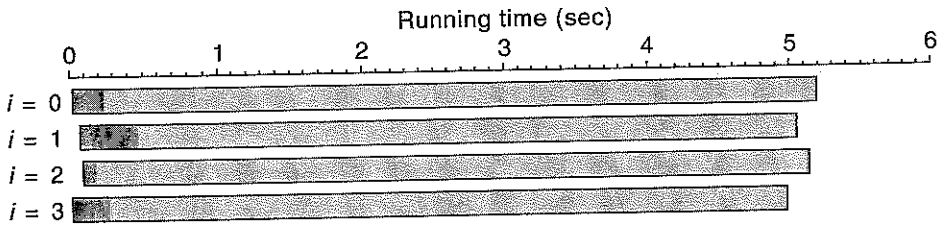


Figure 4.5 Program1Smp execution timeline, regular computer

The reason has to do with how the JVM works. A modern JVM includes a **just-in-time (JIT) compiler** that converts the Java bytecode instructions into native machine code instructions as the program runs. The JVM then executes the machine code directly instead of interpreting the Java bytecode; this greatly increases the program's execution speed. Furthermore, a modern JVM monitors which sections of bytecode are executed most frequently and compiles just those sections to machine code, leaving the remaining sections as interpreted bytecode. This avoids spending the time it would take to compile infrequently used sections of bytecode. (Sun Microsystems refers to this as a **HotSpot JVM**.) However, it takes a certain amount of execution before the JVM detects that the `isPrime()` subroutine is a hot spot and compiles it to machine code. With four threads all calling the subroutine at the same time, the JVM can detect the hot spot, and compile it to machine code, sooner in the parallel version than in the sequential version. This allows more of the parallel version's running time to be executed in the faster machine code mode, thus reducing the parallel version's running time compared to the sequential version. (To verify that this is in fact what's going on, try running both versions with the JIT compiler disabled; the parallel version then invariably takes longer than the sequential version due to the parallel version's extra overhead.) We will see further instances of how the JVM's behavior influences program performance as we study parallel programming in Java.

4.6 The Rest of the Book

Let's step back and look at what we've done. We started with a problem statement. We wrote a sequential program and a parallel program to solve the problem. The parallel program illustrated both general parallel programming techniques (in this case, achieving repetition via multiple threads) and specific Parallel

Java features (parallel team and parallel region). Then we ran the sequential and parallel programs, measured their running times, and gained some insight about parallel programming by comparing the programs' performance.

The rest of the book will be much the same—solving a series of problems that are chosen to illustrate various parallel programming techniques and studying the programs' performance measurements. In Part II, we will begin with SMP parallel programs, because those are quite similar to regular sequential programs. Then, in Part III, we will move on to cluster parallel programs, which are a bit more different from regular sequential programs due to the explicit message passing that is needed. In Part IV, we will combine techniques for SMP parallel programming and techniques for cluster parallel programming to write hybrid parallel programs. While the problems we solve in Parts II through IV will be interesting and perhaps fun, they were chosen solely for pedagogical reasons—to illustrate parallel programming techniques—and are not necessarily problems with any great significance in the real world. Finally, in Part V, we will apply the techniques we've learned to solve some *real-world* problems using parallel computing.

4.7 For Further Information

On the HotSpot JVM, and performance tuning of Java programs in general:

- Steve Wilson and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Addison-Wesley, 2000. Available online at: http://java.sun.com/docs/books/performance/
- Java Performance Documentation. http://java.sun.com/docs/performance/index.html